

# An Introduction to

## Sybase<sup>®</sup> Adaptive Server<sup>®</sup> Enterprise's Modern Optimizer

---

A Collection of Articles Reprinted from ISUG Technical Journal

Eric Miner  
Sr. Engineer  
Sybase, Inc.  
DST Engineering  
Optimizer Group

# Table of Contents

---

Introduction	<i>3 of 31</i>
Introduction to Abstract Query Plans in ASE 12.0	<i>4 of 31</i>
Where, Why and How To Add or Modify Optimizer Statistics	<i>10 of 31</i>
Optimizer Mythology and Folklore (and Some FAQs)	<i>17 of 31</i>
Using Optdiag Simulate In 'What-If' Analysis	<i>22 of 31</i>
An Introduction to Utilizing Optdiag Output	<i>27 of 31</i>

# Introduction

Interest among users about how the ASE optimizer works, and how to work with it has grown considerably in recent years. Performance in general, and optimization in particular are quickly becoming "the last frontier" of the RDBMS.

Recent versions of ASE have brought a great deal of change in the realm of optimization. The five articles in this collection all deal with various aspects of these changes. Some of them give you detailed information on how to take advantage of new functionality in the optimizer while one of them dispels old folklore about the optimizer.

"An Introduction to Utilizing Optdiag Output" – This article describes how to read and understand the output of the optdiag utility. This utility allows you to read, write and simulate the statistics used by the optimizer to determine the most efficient method to retrieve the required data. In "Using Optdiag Simulate in 'What-If' Analysis" you'll find information on how to use Optdiag's simulate mode to simulate changes to your data and perform a 'what-if' analysis.

"Where, Why and How to Add or Modify Optimizer Statistics" give you tips and tricks on how you can add and/or change the optimizer statistics in order to give the optimizer the most accurate information possible about your data.

"An Introduction to Abstract Query Plans in ASE 12.0" introduces a new feature of ASE 12.0, Abstract Plans. This feature allows you to run a query, capture and save the query plan created by the optimizer and then reuse it whenever the same query is run again. You can edit an Abstract Plan, thus telling the optimizer exactly what to do, or simply use it as is. This is a very powerful and flexible feature.

The final article, "Optimizer Mythology and Folklore (and Some FAQs)" dispels some old optimizer related myths and folklore. The more correct information you have about how the optimizer works the easier your tuning jobs will be. There are also some frequently asked optimizer questions.

I hope you find these articles informative and useful in your work with ASE.

*Eric Miner*  
*Sr. Engineer*  
*Sybase, Inc.*  
*DST Engineering*  
*Optimizer Group*

# Introduction to Abstract Query Plans in ASE 12.0

By Eric Miner

*Looking at one of ASE 12.0's most powerful new features*

**A**SE 12.0 introduces a new and powerful feature, Abstract Query Plans (APs). This is a very large feature which cannot be fully covered in one article. However, we will at least scratch the surface to provide an introduction to the value of this new capability.

## What is a Query Plan?

Before we go into APs in detail, let's back up a bit and define query plans and their role in executing a query.

The optimizer's job is to determine the most efficient way (method) to access the data and pass this information on for execution of the query. The query plan is the information on how to execute the query, which is produced by the optimizer. Basically, the query plan contains a series of specific instructions on how to most efficiently retrieve the required data.

Prior to ASE 12.0, the only option has been to view it via showplan or dbcc traceon 310 output (as FINAL PLAN). Once execution is complete, the query plan is gone.

In ASE 12.0, however, APs make each query plan fully available to you. They can be captured, associated to their original query, and reused over and over, bypassing the optimizer fully or partially. They can even be edited and included in a query using the new T-SQL PLAN statement.

You don't have to use APs, of course; in the vast majority of cases the optimizer will indeed choose the most efficient plan.

However, sometimes after an upgrade or in the case of a bug—or the times in which an index strategy was designed for a majority of queries, but a small subset of queries runs less efficiently against these same tables—an efficient plan is not chosen. These are the cases that APs were designed for.

## Why Abstract Plans?

In most cases, changes to the optimizer's cost model result in more efficient query plans. In some cases, of course, there will be no change; and in some there may be performance regressions.

APs can be captured before applying the upgrade, and then, once the upgrade has been completed, queries can be rerun to see if there is any negative change in performance. If a performance regression is identified after the upgrade, the AP from the previous version can then be used in the new version to run the query and Sybase can be contacted regarding the problem. In the case of a possible optimizer bug, you may write an AP to workaround the problem while Sybase investigates it.

In both cases, once the problem is fixed, you can stop using the AP if you choose. It is the possibility of performance regressions between versions of ASE and optimizer-related bugs that APs were first designed to alleviate. These are very good uses for APs, but the power and flexibility that APs provide can be used in other ways too.



*Eric Miner has been with Sybase since 1992, working with the optimizer team in engineering, as well as focusing on optimizer issues for technical and product support engineering. He can be reached at [eric.miner@sybase.com](mailto:eric.miner@sybase.com).*

## What are Abstract Plans?

Put simply, APs are a persistent human-readable and editable copy of the query plan created by the optimizer. APs can be captured when a query is run. Once captured, they can then be associated to their originating query and used to execute that query whenever it is run again.

When ASE creates an AP, it contains all the access methods specified by the optimizer such as how to read the table (table or index scan), which index to use if an index scan is specified, which join type to use if a join is performed, what degree of parallelism, what I/O size, and whether the LRU or MRU strategy is to be utilized.

Let's take a quick look at a couple of simple APs.

```
select * from t1 where c=0
```

The simple search argument query above generates the AP below:

```
(i_scan c_index t1)
(prop t1 (parallel 1) (prefetch 16) (lru))
```

This AP says access table t1 is using index c\_index. It also says that table t1 will be accessed using no parallelism, 16K I/O prefetch, and the LRU strategy.

```
select * from t1, t2
where t1.c = t2.c and t1.c = 0
```

The simple join above results in the AP below:

```
(nl_g_join t1.c = t2.c and t1.c = 0 (i_scan i1 t1) (i_scan i2 t2)
(prop t1 (parallel 1) (prefetch 2) (lru))
(prop t2 (parallel 1) (prefetch 16) (lru))
```

This AP says to perform a nested loop join, using table t1 as the outer table, and to access it using an index i1. Use table t2 as the inner table; access it using index i2. For both tables, use no parallelism and use the LRU strategy. For table t1, use 2K prefetch and for table t2, use 16K prefetch. As you can see, with a little practice APs are easy to read and understand.

## How Do APs Work?

As mentioned earlier, APs can be captured (created) when you run a query and then associated to that query for use when executing it again. Note that in cases of diverse parameters on each execution, each SARG value will result in a separate AP.

## Capture (Dump) Mode: Creating APs

APs are captured when you turn on the capture mode (a.k.a., dump mode) and run a query. Capture mode can be turned on with a session-level **set** command or with a dynamic server-wide configuration option. See the following example.

```
set plan dump on
```

ASE configuration value:

```
sp_configure "abstract plan dump", 1
```

The optimizer will optimize the query as usual, but it will also save a copy of its chosen query plan in the form of an AP. Keep in mind that if you're capturing an AP for a compiled object, such as a stored procedure, you'll need to ensure that it is recompiled. You can do this by executing it with **recompile** or by dropping and recreating it and then executing again.

As APs are captured, they are written to the new system table *sysqueryplans* and placed in a capture group. This group can be the default group, *ap\_stdout*, or a group you've created and specified for AP capture. Groups are created and administered using new system stored procedures provided for working with APs (see sidebar on page 11). When ASE searches for an AP that matches the query, it will check the currently active association group. The association group can be the same as the capture group.

When an AP is captured, it is stored along with a unique AP ID number, a hash key value, the query text (trimmed to remove white space) the user's ID, and the ID of the current AP group. The hash key is a computed number used later to aid when associating the query to an AP, created using the trimmed query text. In general, there are at least one million possible hash key values for every AP, thus making conflicts unlikely.

## Create Plan

APs can also be created manually without conflict by using the new **create plan** T-SQL statement, writing the AP text along with the SQL statement. When you save an AP using the **create plan** command, the query will not be executed. It's advisable to run the query as soon as possible using the AP to ensure that it performs the way you expect it to. Let's take a look:

```
create plan
select c1, c2 from tableA
where c1 = 10
and c2 > 100
"(i_scan tableA_index tableA)"
```

The AP in this example will access tableA using index tableA\_index.

### Associate (Load) Mode: Using APs To Execute Queries

Once captured, an AP can then be associated to the originating query. Multiple users can have multiple, different APs bound to the same SQL statements. When associate mode (a.k.a. load mode) is on, sysqueryplans is checked for APs that are associated to the query. This is done by converting the incoming query to a hashkey value and looking for a matching one, as well as using the userid. If a match is found, the corresponding AP is then used to execute the query. If no AP is found, the optimizer compiles a new query plan which is used to execute the query as usual.

```
set plan load on
```

ASE configuration value:

```
sp_configure "abstract plan load", 1
```

### Replace Mode: Overwriting Existing APs

If you want to replace existing APs, you have to use the *replace mode*. To replace APs, make sure that capture mode is on, then turn replace mode on as well. If you run a query with only capture mode turned on, the resulting AP will not be saved to sysqueryplans. Set command:

```
set plan capture on
set plan replace on
```

ASE configuration value:

```
sp_configure "abstract plan replace", 1
```

When should you use replace mode? If you've made any physical changes to your database (adding an index or table), you should rerun your queries with replace mode on to insure that APs reflect the changes. Since the statistics used by the optimizer are very important, you may want to replace APs after you've made changes via update statistics or with optdiag.

However, if you are using APs to bypass the effects of statistics changes, there is no need to replace them.

## Full Plans and Partial Plans

### Full Plans

When the optimizer creates an AP, it always creates what's called a *full plan*. This is a copy of the complete query plan that the optimizer has chosen. Let's use one of the APs we looked at previously as an example.

```
(nl_g_join t1.c = t2.c and t1.c = 0 (i_scan i1 t1)(i_scan i2 t2)
(prop t1 (parallel 1) (prefetch 2) (lru))
(prop t2 (parallel 1) (prefetch 16) (lru)))
```

The AP above is a full plan because it describes exactly how to execute the query. Thus the optimizer can be completely bypassed when using the AP.

### Partial Plans

*Partial plans* contain some instructions for execution but leave others up to the optimizer. Partial plans are never created by ASE. However, you can create an AP by using the new T-SQL statement:

```
create plan
"select * from t1, t2
wheret1.c = t2.c and t1.c = 0"
"(nl_g_join t1.c = t2.c and t1.c = 0 (i_scan t1) (t_scan t2))"
```

The AP now tells ASE to execute our sample join using nested loop join as before, but now we specify that an index scan on table t1 must be used and that the choice of index is up to the optimizer. For table t2, we are saying that only a table scan can be used to access the table.

## APs vs. the Force Options

In pre-ASE 12.0, the only way users can affect the optimizer's behavior is to use the few force options available (primarily *force index* and *forceplan*). While these options are useful, they are not very flexible for fine-tuning.

With *force index*, you can only force an index (or table scan) to be used—you can't tell the optimizer to use an index but leave the choice of which index open. With *forceplan*, you can only force the join order of the tables to match the order of tables in the **from** clause of the query; you cannot force the type of join.

Another limitation of the force options is that they require you to modify individual queries. Changing the SQL

code is not usually practical or even possible. If you do use the force options, you need to do a lot of testing to make sure that the forces are indeed performing correctly. APs can now be used to execute the query exactly as you specify. As we've seen, they allow users to completely bypass the optimizer, or allow it to make some choices within specific limits.

### The Plan Statement

You may choose to include an AP in a query by using the new T-SQL **plan** statement. The plan statement enables you to execute a query using the AP you specify. Think of it as the "ultimate force option."

```
select t1.c1,t2.c2
from t1,t2
where t1.c1 = 400 and t2.c2 < 100
```

Let's assume that the above query uses nested loop join with table t2 as the outer table, but that we want to see if a sort merge join with table t1 as the outer table would be more efficient. We can use the **plan** statement to specify that a sort merge join is used.

```
select t1.c1,t2.c2
from t1,t2
where t1.c1 = 400 and t2.c2 < 100
plan
"(m_g_join (i_scan i1 t1)(i_scan i2 t2))"
```

Here we tell ASE to execute the query as a sort merge join, but specify that table t1 be the outer table. We're also specifying which indexes to use, which we could not do before.

### The Cost Of Using APs

As with everything in life, using APs has its cost.

As APs are captured, there will be a write to sysqueryplans for every one captured. Each write of an AP will also be logged. It is advisable to capture your APs on a test ASE or when usage of your production ASE is minimal. With *association mode* on, sysqueryplans has to be searched for an AP that matches the query. In general, this "lookup overhead" is well compensated for by the use of APs.

With *association mode* on, a per-connection "Abstract Plan cache" is created. If there are 20 or less APs in the group, their hashkeys will be stored in the AP cache. When a query is run with *association mode* on, the cache will be checked. If there is no matching hashkey in the cache, there is no matching AP for the query. The number of 20 APs was

created to limit the amount of PSS structure memory the cache uses: The AP cache memory is taken from main ASE memory.

### How To Tell When APs Have Been Used To Execute a Query

You can verify that an AP has been used to execute a query by looking at a couple of outputs, showplan and traceon 302. Showplan is the easiest way to verify that an AP was used to execute a query. Immediately following the first line of output, you'll see a message indicating that an AP was used. The AP ID will be included (in this case ID number 608005197):

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using an Abstract Plan (ID : 608005197).
```

For those who use traceon 302 output, this too would give indications that an AP has been used, although they are not quite as easy to read:

```
For table 'T1':
User forces index 2 (index name = I1)
User forces index prefetch size of 2K.
User forces data prefetch size of 2K.
User forces LRU buffer replacement strategy on index and data pages
User forces parallel strategy. Parallel Degree = 20
```

These are the same messages that 302 would report if you forced each of these items. If you see a list of forced options in traceon 302 output, and you haven't set these force options on, you know an AP has been used. Keep in mind that if you use a partial plan AP, the force options list will only contain those methods the AP specified for execution of the query. Methods chosen by the optimizer will not be included in this list.

### APs and Upgrading ASE

Let's take a look at a couple of basic upgrade scenarios.

#### Upgrading to ASE 12.0 from an Earlier Version

In this case, it's impossible to have captured APs from the earlier ASE version. Therefore, you'll need to run performance tests as soon as possible after upgrading to ASE 12.0. If you find any performance regressions, rerun the query or queries to capture APs. If upon examination they show any identifiable problems, edit the AP or create a partial plan to work around them, and then use the AP whenever the query

is run. As usual, if there is any performance regression after an upgrade, you should contact Sybase as soon as possible.

### Upgrading from ASE 12.0 to a Later Version

Now let's assume you are upgrading from ASE 12.0 to a later version. In this case, you need to turn on the AP capture mode and run your queries before the upgrade. Save these APs off to a separate group, and perform the upgrade.

Once complete, rerun your queries and check for any performance problems. If there are none, that's as far as you need to go. However, if any query shows a performance regression, turn on AP capture and save the inefficient AP. Then find the efficient AP in the pre-upgrade group, copy it over to the post-upgrade group, and use it to execute the query. You'll want to report the problem to Sybase and send them the inefficient AP.

### Limitations of APs

APs do have a couple of issues that can be seen as limitations. The first is the fact that the userid of the user who created the AP can be used as part of the check when association mode is on. In other words, only the original user can use the AP to associate to a query. However, the AP can simply be copied to a group available to another userid, thus removing the issue but adding an extra step in the process.

Another issue with APs is their use with ad-hoc queries. Since the exact text of the query is stored with the AP, and is used to associate to its originating query, any changes in the text will cause it to no longer match the one stored with the AP. Let's take a quick look at a couple of examples. The query below is used to capture an AP:

```
select col1,col2 from t1 where col1 = 10 and col2 > 20
```

The query below is then run:

```
select col1,col2 from t1 where col1 = 11 and col2 > 20
```

The query is no longer the same as the original query because the search value on column c1 is different. The AP cannot be associated with it.

Another difference in queries that can disqualify the use of an AP is "white space" in the query when used in association.

```
select col1,col2 from t1 where col1 =10 and col2 > 20
```

The query above cannot be associated to the AP because the equality operator has no space separating it from the search value, as it does in the original query. These issues may make APs difficult to use with ad-hoc queries.

### A Good Time to Use An AP

A situation in which APs may prove very useful is in joins of many tables. In ASE 12.0, the number of tables that can be referenced in a query has increased to 50. This 50-table limit was designed primarily to facilitate the execution of queries with a large number of "non-flattened" subqueries. However, it is now possible to also use more than 16 tables in a join. If you decide to write joins containing many tables you might want to consider using APs to execute them.

Here's why. As the number of tables on the **from** clause of a join increases, so does the number of possible join orders for which the optimizer has to perform estimates. Add qualifying indexes to that, and the number of possible query plans to be costed grows exponentially. As the number of plans grows, so does the time to optimize the query.

This is where APs come in. If you have a commonly run join that contains a large number of tables and you've found that the compile time is very long, you might want to capture and associate an AP to the join. The AP will bypass the optimizer's need to cost all those plans.

### A Useful AP Trick

Many users have asked how to tell whether and how often an index is used. At best, this is difficult to do without APs. Using APs, the following steps can be used to get this information:

1. Turn on AP capture
2. Run the queries that you're interested in examining
3. Once capturing the APs is done export them to a user table using `sp_export_qpgroup` (see sidebar).
4. Create an index on the column named text to speed up your queries.
5. Query the user table using the index name to count the number of time the index was used in the set of queries you ran. For example:

```
select count(text) from my_ap_table where text like
"%index_name%"
```

This will give you a count of how many times the index name appears in APs in the capture group. Thus, you'll know how many times the optimizer has chosen to use that index. It's not eloquent, but it's useful.

## Conclusion

APs provide powerful new methods to enable users to bypass the optimizer's decision process partially or completely. They also provide a very granular and flexible way to control the execution of your queries. Finally, APs can act as a workaround to optimizer-related performance regressions while Sybase looks into the problem.

A final note: You can develop applications and include APs for use in a production environment. However, it is highly advisable to capture APs for the application from datasets that will be very close to the size of the production datasets. It may not be easy to test against large datasets. In such cases, you can use `optdiag simulate` to simulate large datasets for the capture of APs. ■

## System Stored Procedures For AP Management

ASE 12.0 provides 15 new system stored procedures to help you with managing APs. These stored procedures are divided into fifteen functions.

### *AP plan groups:*

- ◆ **sp\_add\_qpgroup:** Adds an AP group
- ◆ **sp\_drop\_qpgroup:** Drops an AP group
- ◆ **sp\_help\_qpgroup:** Provides information about the group and its APs
- ◆ **sp\_rename\_qpgroup:** Used to rename a group; useful when you want to make a group and its APs available to someone other than the user who created them

### *Find an AP:*

- ◆ **sp\_find\_qplan:** Used to locate a plan by searching either the query text or the AP text

### *Working with individual APs:*

- ◆ **sp\_help\_qplan:** Provides information about a query plan including its hashkey, id number, the query text and plan text
- ◆ **sp\_copy\_qplan:** Used to copy an AP from one group to another using its id number
- ◆ **sp\_drop\_qplan:** Used to drop an individual AP based on its ID number
- ◆ **sp\_cmp\_qplans:** Used to compare two APs by the ID numbers; this procedure shows the differences between queries and plans stored for each of the APs
- ◆ **sp\_set\_qplan:** Used to write plan text or edit an existing plan

### *Working with all plans in a group:*

- ◆ **sp\_copy\_all\_qplans:** Used to copy all APs from one group to another; useful when you want to put APs into a group or groups accessible to users with a different userid than the userid that captured the APs.
- ◆ **sp\_cmp\_all\_qplans:** Used to compare APs in any two groups. Provides information on the number of APs that are the same in the groups, the number of plans that have the same hashkey but different plans, and the number of plans that are present in one group but not the other. Useful when you want to make sure that another group contains all the plans you need.
- ◆ **sp\_drop\_all\_qplans:** Used to drop all APs in a specified group

### *Importing and exporting (copying APs to a user table):*

- ◆ **sp\_export\_qpgroup:** Used to copy the APs in a specified group to a user table with `select into` to create the target user table. The structure of the user table is the same as `sysqueryplans`
- ◆ **sp\_import\_qpgroup:** Used to copy plans from user tables created with `sp_export_qpgroups` into `sysqueryplans`. This can be used if you edit APs in the user table.

As you can see, APs can be used to perform many administrative chores. ■

# Where, Why and How To Add or Modify Optimizer Statistics

By Eric Miner

*Supporting better decision –  
making based on optimizer  
statistics*



*Eric Miner has been with Sybase since 1992, working with the optimizer team in engineering, as well as focusing on optimizer issues for technical and product support engineering. He can be reached at [eric.miner@sybase.com](mailto:eric.miner@sybase.com).*

In previous issues of this journal, we've discussed the new optimizer statistics introduced in ASE 11.9.2 and how to read, write, and simulate them using optdiag. This article discusses why, where, and how you might want to add or change optimizer statistics. These statistics are used by the optimizer to estimate the most efficient way to access the data required by a query. They are the only information the optimizer has about your tables, indexes, and data. This article will provide information you can use to help make decisions on how to take advantage of the powerful and flexible changes to optimizer statistics in ASE 11.9.2 and above.

## ASE's Optimizer Statistics

As of ASE 11.9.2 the optimizer statistics are stored in two system tables. *Systatistics* stores the column level statistics that were previously stored in the single distribution page (a.k.a., the distribution statistics). The date and time of the last modification of the statistics are stored here. The statistics stored in *systatistics* can be written directly, as discussed below.

*Systabstats* contains the table/index level statistics such as row count, page count, index height, and the cluster ratios. These statistics are maintained dynamically; changes are first applied in memory and then flushed to *systabstats*. It's this regular overwriting of the table/index level statistics that makes it futile to attempt to modify them. In fact, *optdiag* will not allow you to input a file with modifications to the table/index statistics.

Before we continue, keep in mind that adding and/or modifying statistics is not absolutely necessary. However, it is highly recommended that you consider taking advantage of the flexibility and power of the new statistics and test their use at your site. You may find that adding or modifying the statistics dramatically increases the efficiency of your query plans. On the other hand, you may find for your datasets that keeping the statistics as they were works just fine.

## Some General Terminology

Let's give some definitions before going on.

- ◆ **Histogram:** Used to represent the distribution of values in a column. Used by the optimizer when costing SARGs. Consists of steps/cells which represent either a range of values or a single value, weights which represent the percentage of the column occupied by the values of the cell and the boundary values which delineate the cell.
- ◆ **Attribute of an index:** Any column that is part of an index, single or multi-column.
- ◆ **Column level statistics:** Statistics that describe the data in a column.
- ◆ **Table/index level statistics:** Statistics that describe a table and its indexes.

## Natural, Default, Added and Modified Statistics

Column level statistics are created/updated based on the underlying data. This happens when you run any form of update statistics, and can be thought of as natural statistics.

The default statistics can be thought of as the result of running:

```
update statistics table_name [index_name]
```

This will create or update statistics on the leading column of the index/indexes. This was the only option in pre-ASE 11.9.2 versions. When you use **update statistics** to create/update statistics on any column other than the leading column of an index or indexes, the resulting statistics are called *added* or *additional* statistics.

Any statistics value that you directly change via `optdiag` is considered a *modified* statistic.

### Why Add or Modify Statistics?

Adding or modifying statistics can result in more efficient query plans than in previous versions of ASE. This is primarily due to the fact that you can now provide the optimizer with a great deal more information about your indexes and data.

#### Adding Statistics: Why, Where, and How

Adding statistics to columns, indexed or non-indexed, is the most common way to take advantage of the new statistics. Let's look at adding statistics to inner columns (minor attributes) of composite indexes first.

With statistics only on the leading column (major attribute) of an index, the optimizer is limited to information for that column only, and must make assumptions about any other columns in the index. The additional statistics on inner columns gives the optimizer a complete view of the index, and it will not need to make any assumptions about how selective the column is. Let's take a look at the example of a simple test query:

```
select * from test
where col1 > 200
and col3 = .40
and col2 <= 300
```

In the first example, statistics are only on the leading column of the index (the default statistics). You can see that without statistics on col2 and col3, the optimizer has to use default

selectivity values (formerly called *magic numbers*) to estimate selectivity for those columns. These values are assumptions about selectivity and are not likely to be accurate.

traceon 302 output:

```
Estimated selectivity for col1,
selectivity = 0.999597.
```

```
No statistics available for col2,
using the default range selectivity to estimate selectivity.
```

```
Estimated selectivity for col2,
selectivity = 0.330000.
```

```
No statistics available for col3,
using the default equality selectivity to estimate selectivity.
```

```
Estimated selectivity for col3,
selectivity = 0.100000.
```

```
costing 22243 pages, with an estimate of 16493 rows
Search argument selectivity is 0.032987.
```

Now we add statistics to all columns of the index (col1, col2 and col3):

```
Estimated selectivity for col1,
selectivity = 0.999597.
```

```
Estimated selectivity for col2,
selectivity = 0.014925, upper limit = 0.052684.
```

```
Estimated selectivity for col3,
selectivity = 0.020003, upper limit = 0.060138.
```

```
costing 5898 pages, with an estimate of 149 rows
Search argument selectivity is 0.000298.
```

Statistics io output:

```
Table: test scan count 1, logical reads: (regular=5895 apf=0
total=5895),
physical reads: (regular=143 apf=0 total=143), apf IOs used=0
Total writes for this command: 0

(147 rows affected)
```

After adding statistics to the inner columns of the index, the estimated cost of the query is far more accurate.

Now let's take a look at the effects of adding statistics to a non-indexed column that's participating in a simple join:

```
select * from test t, test2 t2
where t.col1 = t2.col2
and t.l_orderkey > 200
and t.col2 = 100
and t.col1 <= 300
```

Without statistics on test2.col1:

```
Traceon 302 output
Estimated selectivity for col1,
selectivity = 0.100000.
```

With statistics on test2.col1:

```
Estimated selectivity for col1,
selectivity = 0.000052, upper limit = 0.052684.
```

Statistics on the non-indexed column t2.col1 resulted in more accurate cost estimates. In the case of this simple join, the presence of statistics on the non-indexed column resulted in a different join order being used. Having statistics on columns that participate in joins is especially useful when you enable sort-merge joins in ASE 12.0 or above.

### How To Add Statistics To Columns, Indexed or Non-Indexed

The fundamental changes to **update statistics** deal with adding statistics to a column in a number of ways. Also included in the new functionality is the ability to specify the number of steps to use for the column's histogram (distribution statistics). See Sybase documentation for a general introduction to the syntax and the basic changes to update statistics. However, below is some information on which form of **update statistics** to run in various situations.

To add statistics to columns, one column at a time, use the following syntax:

```
update statistics table_name (column_name)
```

You may have heard or read about **update index statistics**.

This extension will create or update statistics for all columns of all indexes of a table or for a specified index:

```
update index statistics table_name [index_name]
```

Another option is **update all statistics**:

```
update all statistics table_name
```

This option will create or update statistics on all columns of a table. Be very cautious of this option for a couple of reasons: it can take a very long time to run and in the vast majority of cases having statistics on all columns of a table is not necessary. Always test changes to the statistics completely before implementing them in production.

### A Word On Maintaining Column-Level Statistics

As with everything in life, there's a trade-off with adding statistics. As we all know **update statistics** takes time to run, and the larger the table the longer it takes.

In pre-ASE 11.9.2 versions, **update statistics** had to read the leading column of an index and gather the statistics for that column only. Since the column was in sorted order, there was no need to do anything but the read (scan). In 11.9.2 and above, you can now add statistics to any column. If the column is not the leading column of an index, it needs to be sorted before the read in order to gather the statistics. This adds I/O and time to the process. It will also require space in tempdb to handle the worktable for the sort. If the column is in an index, the size of the worktable will be the size of the index leaf pages, plus or minus a few pages. If the column is not in an index, the worktable will be the size of the table.

In 11.9.2 or above, an additional scan will need to be done to gather the cluster ratio statistics. A table scan will need to be done if you specify only a table name or a table and column name in update statistics. If you specify an index, an index leaf scan will occur; except in the case of a clustered index on an All Pages Locked table, where a table scan will be done.

In most cases, the cost of maintenance will be outweighed by the more efficient plans the optimizer will generate. The added maintenance cost is another good reason to test the

effectiveness of adding statistics to columns. This added maintenance cost is also a very good reason to rethink running `update all statistics` or `update index statistics` without testing first.

### Changing the Number of Requested Steps

The number of steps (cells) in the histogram of a column has a direct effect on the optimizer. If you create an index or statistics on a column that has no statistics, the default number of steps (20) is used. If statistics exist on the column, the number of requested steps used will be the last number of requested cells used, unless you specify a different value. You can also specify the number of steps to be used. There are a couple of reasons you may want to consider changing the number of steps for a column.

The first is to get more Frequency Count cells in the histogram. These are the most accurate type of cell, since they represent only one value. Frequency Count cells generally occur when a value or values occupies a large number of rows. A frequency count cell can be “pulled” from a Range Cell (a cell representing multiple values) if a value occupies more than 50% of a cell width. The width of a cell is the number

of rows divided by the number of requested steps minus 1. You can use this formula to tell if a value that you know to be highly duplicated in your dataset will have a frequency count.

Another reason to increase the number of requested steps is to make the column’s histogram more granular. A more granular histogram makes it easier for the optimizer to accurately estimate the cost of a SARG. This is especially true for range SARGs. For such SARGs, the optimizer estimates how close a SARG value falls to either boundary of a cell and uses this to estimate selectivity; the narrower the cells, the more accurate this is. Keep in mind that this applies only to Range Cells, since they represent more than one value and a SARG value’s position within the cell must be estimated. When a SARG value falls into a frequency count cell, no estimations are needed since the cell represents only one value. Because the number of steps to use in a histogram is taken from the old distribution page during upgrade, do not delete statistics after upgrade completes.

`Update statistics` has been extended to allow you to specify the number of steps to use for building the histogram(s). For example:

```
update statistics table_name using 100 values
```

The above syntax will update statistics on the leading columns of all indexes in the table and will use 100 as the number of requested steps.

```
update statistics table_name (col1) using 100 values
```

The above syntax will create or update statistics on the specified column, col1, using 100 as the requested number of steps.

The number of steps can also be specified in create index with the same using X values syntax.

There is a slight trade-off when increasing the number of steps. Steps require memory taken from procedure cache whenever the optimizer needs to read them. The amount of memory is small, but the larger the datatype of the column and the more steps in the histogram, the more is needed. In the most extreme cases, this can have an adverse effect on parse and compile time. In most cases, the more efficient query plans that result from more steps far outweighs the cost of caching the steps.

### A Word On Statistics and Upgrade

When upgrading to 11.9.2 or above from an earlier version, you’ll need to take the statistics into consideration.

During upgrade, if a column is the leading column of an index, the old distribution page is read and its values are used to establish the new statistics. This essentially copies the old statistics values into the new values. This copy is not as accurate, as statistics obtained from the underlying data by ASE 11.9.2 or above. For example, the values of the steps in the distribution page are used to create the boundary values for the new histogram steps. However, the weights are estimated by using the number of values that fall between each step of the distribution page. This is not as accurate as getting the boundary values and weights from the data. Frequency count cells are created when a value appears in more than one step

in the distribution page. The number of requested steps to use in the new histogram is the number of steps in the old distribution page.

After upgrade completes, it is highly recommended that you run **update statistics** for all tables. This will gather the new, more accurate statistics. For this first post-upgrade run, use the syntax you used in previous versions:

```
update statistics table_name [index_name]
```

You may have heard that you ought to run **update index statistics**, or even the extreme **update all statistics** after upgrading. While it's recommended that you consider and test the effects of statistics on inner columns of composite indexes and/or on non-indexed columns, you ought to do so after upgrading the statistics to the new format. Hold off adding statistics until you have a chance to test them.

There has been some folklore recently about deleting statistics after upgrade and then running **update statistics**. This is not necessary and can result in inefficient query plans. Since the number of requested steps are taken from the number of steps on the distribution page, the resulting histogram will closely match that of the distribution page. If you drop and recreate the statistics, the histogram will be built using the default of 20 steps. This is because the previous step count is lost when the statistics are deleted. You should avoid changing the granularity of the histogram until you've had a chance to test it against your queries.

The bottom line is that after upgrading to 11.9.2 or above from a previous ASE, do not delete or add statistics until you've tested their effects on your queries. But, do run **update statistics** as soon as possible after the upgrade completes.

### Modifying Statistics: Why, Where and How?

You can now directly write the column level statistics. You can use `optdiag` to do this since it's not advisable, nor supported by Sybase, to write directly to the system tables. There are a number of reasons you may want to write the statistics directly.

First, though, let's talk about how to write the statistics. As mentioned, use `optdiag` to do this by getting an `optdiag` output file. Use the `-o file_name` option at the command line.

Once you have the file, you can begin to modify the statistics. You can use any text editor to do the job. Save a copy of the original `optdiag` output file in case you need to start over. Once the file is edited, you can read it back in via `optdiag` using the `-i` option. When the optimizer uses statistics that have been modified, `traceon 302` will print this message:

```
Statistics for this column have been edited.
```

This will help you know when modified statistics are used by the optimizer.

### A Note On Maintaining Modified Statistics

Let's take a look at what you'll need to do to maintain modified statistics should you decide to use them. The majority of the column-level statistics are not persistent; they will be overwritten the next time you run **update statistics**. The one exception is the default selectivity values. Keep this in mind if you decide to modify statistics, as it will add a step when updating the statistics.

In the most common scenario, you will be modifying only one or a few values. In these cases, you'll need to get a new `optdiag` output file after `update statistics` is run, re-edit it, and read it back in.

### Modifying Statistics

There are a few different situations in which you may want to consider modifying the statistics.

#### Changing the Total Density Value to Deal with Data Skew

Data skew occurs when a few values each occupy many rows of a column, while many values occupy a few rows each. Data skew will have a direct effect on the Total Density value and thus on the optimizer's costing of joins. You'll see a few Frequency Count cells in the histogram surrounded by Range Cells. The effect this has on the Total Density value can result in an inaccurate number of rows being estimated for a join.

How do we modify Total Density to get around data skew? One approach is to set the Total Density value to match the Range Cell density value. This is a fine approach; however, you want to be cautious about setting the Total Density value

too low, since it may result in the optimizer being overly optimistic about the cost of a join on the column, which in turn results in an inaccurate number of rows being estimated. In most cases of data skew, the Range Cell Density is much lower than the Total Density. Optdiag output:

```

Range cell density: 0.0000502421670203
Total density: 0.2697381850000000
Step  Weight          Value
-----
1  0.00000000  <=  0
2  0.05263000  <= 5222
3  0.05264000  <= 10564
4  0.05265000  <= 15779
5  0.05263000  <= 20998
6  0.04125000  <= 24999
7  0.00000000  <  40000
8  0.31933998  =  40000
9  0.05263000  <= 96995
    
```

In the example above, the histogram is skewed on the value 40000 (a Frequency Count Cell) with almost 32% of the column occupied by that value. Notice that the Total Density is much larger than the Range Cell Density. Since the Total Density value is used to cost all joins that the column participates in, setting it too low may adversely effect other joins. Whenever you add **of modify** statistics, make sure to test the changes completely.

A new system stored procedure, *sp\_modifystats*, can be used to set the Total Density value to equal the Range Cell Density value. This procedure is available in ASE 12.0 and 11.9.2.2. Again, be careful not to set the Total Density value too low. If the Range Cell Density is very low, don't use *sp\_modifystats*, but rather change the Total Density directly via *optdiag*.

The Total Density value is also used as the default selectivity value for equi-SARGs when the value is unknown at runtime. If you have queries that have unknown equi-SARG values (usually due to local variables) and the Total Density is not resulting in a good plan, try changing it. Again, keep in mind that any change to the Total Density will effect all joins on the column.

### Changing the Default Selectivity Values

The default selectivity values are used to determine selectivity when the value of a SARG is unknown at runtime. Unknown values usually occur when a local variable is used in a query or a stored procedure (values are known if parameters are used in a stored procedure):

```

declare @1 int
select @1 = 100
select * from table where column = @1
    
```

Unknown values can also result, in some cases, when using functions or expressions in a SARG. In earlier versions of ASE, the magic numbers were hard-coded and could not be changed. In ASE 11.9.2 and above, the default selectivity values can be modified.

There are two default selectivity values. The "Range selectivity" value is used when the value of a range SARG (<, <=, >, >=) is unknown. By default this value is 0.33 (examples are from *optdiag* output):

```

Range selectivity:      default used (0.33)
    
```

The "In between selectivity" is used when the value of a between SARG is unknown. Its default value is 0.25:

```

In between selectivity:  default used (0.25)
    
```

To modify either value make sure your changes are in the format below:

```

In between selectivity:  0.10
    
```

As mentioned earlier, the default selectivity value for an unknown equi-SARG value is the Total Density.

### Modifying Statistics For Out of Range SARG Values

If a search argument value is larger than the largest boundary value in the histogram, or if it is smaller than the smallest boundary value in the histogram, it will be out of bounds. This

can occur if the search value is not in the column; or it is in the column but the statistics are not up to date and it is thus beyond the boundaries of the histogram. In either case, the optimizer must use special costing for out-of-bounds SARG values. The selectivity value used will be either 0.0 or 1.0, depending on the type of operator and where the value falls.

For example, if the value is less than the smallest value in the histogram and the operator is  $\geq$ , the selectivity used will be 1.0. If the value is greater than the largest value in the histogram and the operator is  $\geq$ , the selectivity used will be 0.0. If the value is outside either boundary, and an equi-SARG is being used, the selectivity value will be 0.0. In any case, the selectivity value is likely to not be accurate.

Traceon 302 will print the following message when there's an out-of-bounds SARG value (again, the selectivity value can be either 1 or 0):

```
Estimated selectivity for colA,
selectivity = 0.000000, upper limit = 0.000000.
Lower bound search value 10000 is greater than the largest
value in sysstatistics for this column.
```

There are two ways to get around out-of-bounds costing. The first is to add a dummy row to the table that contains a value well outside a boundary. This value will appear as the histogram value for the lowest or highest boundary of the histogram, the SARG value will now fall within the bounds of the histogram, and costing can be done. This approach is not always acceptable, though, since it requires the dummy data to be in the table.

Another approach is to use `optdiag` to change the value of the lowest or highest boundary value so that the search value always falls within bounds. In the first case, the dummy value is persistent; it will always be in the table. In the second case, the value is not persistent and an `optdiag` output file will need to be edited and read back in after each `update statistics` run.

Here's an example. Let's say it's July 15 and `update statistics` hasn't been run in a while. You want to see rows for July 12. The value July 12 will be out of range. If you were to use `optdiag` to change the last cell value (in this case, step 20) to say August 1, full costing could be done (example from `optdiag` output):

```
18 0.05301946 <= "May 1 2000 12:00AM"
19 0.05290456 <= "Jun 1 2000 12:00AM"
20 0.04818739 <= "Jul 1 2000 12:00AM"
```

### Modifying Histogram Cells

There are a few reasons why you might want to modify the cells of a histogram: If you want to add a dummy value for out-of-bounds SARG values as mentioned earlier; or if you want to add frequency count cells to the histogram. If you want to directly write statistics instead of running `update statistics`, or if you decide to modify the histogram's cells, there are a few rules to keep in mind:

- ◆ The step numbers must increase monotonically
- ◆ The weight of each cell should be between 0 and 1.0
- ◆ The sum of the cell weights must be close to 1.0 (0.99 to 1.01)

If you plan to modify or create a histogram for a column, make sure you understand the distribution of the values in the column. Care must be taken when doing this: Remember the optimizer is dependent on the histogram for costing SARGs. As with any change to the statistics, make sure to test thoroughly before implementing.

### Conclusion

The changes to how optimizer statistics are stored and used provide you with powerful and flexible features and functionality. You can add column level statistics to inner columns of composite indexes, giving the optimizer much more accurate information about an index to use when costing a query. You can add statistics to columns that are non-indexed, thus giving the optimizer a clear view of the column when costing a join. You can specify the number of histogram steps to be used for the column—`update statistics` has been extended to allow you to do this. You can also modify all the column level statistics via `optdiag`, thus changing everything from the default selectivity values to the histogram.

All this power and flexibility requires that you weigh the cost against the benefits and fully test any changes to or additions of statistics. We certainly recommend that you consider taking advantage of this new functionality. ■

# Optimizer Mythology and Folklore (and Some FAQs)

By Eric Miner

*Dispelling the fables that lurk in the darkest areas of ASE optimizer*



Eric Miner has been with Sybase since 1992, working with the optimizer team in engineering, as well as focusing on optimizer issues for technical and product support engineering. He can be reached at [eric.miner@sybase.com](mailto:eric.miner@sybase.com).

Now that we've entered the 21st century, let's take a look at some leftovers from the 20th. We've discussed a number of aspects of the optimizer over the past year. Let's take a look at some shorter but equally important topics.

Over the years, mythology and folklore have developed around a number of areas of ASE. One area that has more than its share of these myths is the optimizer. Strange stories spread far and wide, sometimes very quickly. Sometimes they get repeated so many times they're unquestionably accepted as truth. Sometimes they're even put into Sybase training materials. In this article, I'll try to set the record straight on the more common ones.

Why discuss myths and folklore? They need to be explained in order to provide you with a better understanding of how the optimizer works and its true behavior. This article also includes answers to some optimizer-related FAQs.

## Myth #1: The 20% Rule

*"If more than 20% of a table is returned by a query, the only choice the optimizer has is to table scan."*

This is a very old myth. As the story goes, many years ago a senior engineer answered a question at an ISUG conference. The question itself is lost in the mists of time. The off-the-cuff answer was something like, "Hmm, well, if about 20% of a table (read column) will be returned by a query, a table scan will be done. The percentage varies at times, but it is always around 20%." This myth was picked up quickly. It was quoted by Technical Support, consul-

tants, and even put into the manuals.

Since we don't know the exact question that was asked, it's hard to know exactly what the former engineer was referring to. However, an educated guess would be that he was talking about the pessimistic costing of non-clustered indexes in pre-11.9.2 versions of ASE. Basically, this costing assumed that every read from the leaf of the index to a datapage would cost one I/O. In other words, for every qualifying row in the index, one I/O would be done. This, of course, was not the best assumption to make because it didn't take into account that more than one qualifying row may exist on the same datapage.

Even if he was talking about the pessimistic non-clustered index costing, the "20% Rule" still does not hold true. The optimizer's decision of which access to use in order to get the required rows is based on the estimated cost, not on a set percentage of the column. The existence of covering indexes is one way to disprove the "rule." Another is the fact that a clustered index can easily be chosen to return values based on SARGs and joins of columns.

In 11.9.X and above, the Data Row Cluster Ratio is used to measure how well clustered a column's rows are in relation to the data pages. This value is used when estimating the cost of using a non-clustered index. With it, the optimizer can estimate how many qualifying rows can be read in a single I/O.

While the "20% rule" may have seemed to be a reasonable rule of thumb at the time it began, it has never accurately described the optimizer's behavior.

**Myth #2: Update Statistics Improves Performance**

*“Update statistics will give you good performance.”*

**Update statistics** is only guaranteed to result in statistics that are up-to-date as of the end of its run. It does not guarantee good performance. For example, if the values of a column are dense (a gender column is an extreme example), **update statistics** can't change that fact. It can only record the data distribution it sees.

The optimizer depends on the statistics as the only view of its universe. As the statistics change, so can the optimizer's view. Sometimes the change is fine and plans are very good, sometimes not. It's possible that you'll see plans change drastically after **update statistics** is run because the optimizer's view of the data has changes a great deal. This is why it's a good idea to keep a copy of the statistics in the form of an `optdiag` output file, just in case. It all depends on your dataset, how it changes and what your queries are doing.

**Myth #3: When to Run Update Statistics**

*“Update statistics should be run whenever the data changes by 5-10%.”*

This is not necessarily right or wrong. In general, it's a fine guideline. How often to run **update statistics** is completely dependent on changes to the distribution of values in your dataset, the work being done, and the efficiency of the query plans generated by the optimizer. Some datasets whose distribution changes regularly may need to have statistics updated very often, others less often. And, some datasets may never need to have **update statistics** run in order for the optimizer to generate the most efficient query plans. You need to test and monitor performance in order to determine the optimal intervals in which to run **update statistics**.

A rule of thumb such as this isn't harmful, but you may be doing more maintenance work than is necessary.

**Myth #4: Update Statistics Does Cleanup?**

*“You should always use update all statistics” or “Update all statistics will ‘clean up’ all your statistics.”*

The **update all statistics** command will gather or update statistics on all columns of the specified table. It will also update partition statistics.

It is not likely that you'll need statistics on all columns of a table. You should conduct complete tests to determine if statistics on all columns of a table are necessary. Adding statistics to columns that previously didn't have them is likely to result in different query plans. You need to verify that the additional statistics are helping the optimizer produce more efficient plans. While it's recommended that you consider

adding statistics to inner columns of composite indexes and/or to non-indexed columns, always run tests before putting any new **update statistics** syntax use into production.

Since **update all statistics** must gather statistics on all columns of the table, it will add a great deal of time to your statistics maintenance. If you want to update your partition statistics, use **update partition statistics**.

**Myth #5: Deleting Statistics after Upgrades**

*“You should delete statistics after upgrading to 11.9.X (or above) from an earlier version.”*

This is a relatively new myth. It began as a solution to a “print bug” in early 11.9.2, which simply resulted in annoying messages in `optdiag` and `trace 302` output saying that the column's statistics were obtained from upgrade. The message was inaccurate because **update statistics** had been run in the newer ASE and the statistics were in the new format. It was found that if the column's statistics were deleted and **update statistics** rerun, the message went away.

The problem with deleting the statistics and rebuilding them is that you lose the number of requested steps that was inherited from the old distribution page. After deleting the statistics, if you don't specify a number of steps to use, the default of 20 steps is be used when rebuilding. This could result in Frequency count cells not appearing in the histogram because the cells are now too wide for highly duplicated values to get a cell of their own. Also, if there are range SARGs (<, <=, >, >=), the smaller number of steps may change the optimizer's cost estimates, because the cells are now wider.

Let's take a quick look at the sample histograms below. Let's say that the old distribution page had 200 steps for this column. That number is then used as the “requested steps” during upgrade and is marked as the step count for the column. Let's suppose we have a search clause in a query:

```
where column between 'July 1, 1996' and 'August 1, 1996'
```

This range of values would fall into cell (step) 4 of the histogram below (a cell is inclusive of its upper bound value and exclusive of its lower). The optimizer would then need to estimate how much of the cell would qualify in order to estimate the cost. When the histogram has 200 cells, they are approximately ten times narrower than when the histogram contains only 20. It will be easier for the optimizer to determine how close the search clause values are to the boundaries of the cell and thus how much of the cell's weight to use in the cost estimate.

200 steps (column contains a little over 300,000 rows):

Step	Weight	Value
1	0.00000000	<= "May 29 1992 11:53:32:996AM"
2	0.00510867	<= "Oct 20 1995 9:58:24:000AM"
3	0.00517948	<= "May 7 1996 8:22:57:000AM"
4	<b>0.00516177</b>	<= <b>"Oct 17 1996 12:32:37:000AM"</b>
5	0.00514407	<= "Jan 6 1997 10:44:42:000AM"

Default 20 steps:

Step	Weight	Value
1	0.00000000	<= "May 29 1992 11:53:32:996AM"
2	<b>0.05263028</b>	<= <b>"Nov 18 1997 5:11:39:000PM"</b>
3	0.05275065	<= "Aug 6 1998 12:40:39:000PM"
4	0.05264090	<= "Feb 26 1999 10:33:32:000AM"
5	0.05263381	<= "Dec 6 1999 5:37:33:000AM"

The best thing to do after upgrading to 11.9.X or above is to run **update statistics** as you did in the earlier version and then test a set of queries. In the majority of cases, there is no reason to delete the statistics or change the number of steps. See my Q1 2000 *ISUG Technical Journal* article “An Introduction to Utilizing Optdiag Output” for more information on the different types of histogram cell, how they’re used, and their advantages.

#### Myth #6: Trace Flag Sets Optimizer Behavior

*“There’s a trace flag that will set the behavior of the optimizer back to that of (insert previous ASE version number).”*

This folklore has been around for a long time. There is no single trace flag that will set all of the optimizer’s behavior back to that of an earlier version, and there never has been one. Over time, a number of trace flags that effect individual optimizer behaviors have been introduced, primarily to address specific bug situations. Sometimes traces are removed in a new version of ASE; sometimes they stay around for many versions. They are usually not supported or documented in any official manner. I will discuss some of these optimizer related trace flags, in detail, in a future article.

#### Myth #7: Too Many Steps Hurt Performance

*“Don’t use too many steps when creating an index or updating statistics, because it will hurt performance.”*

This myth comes from the fact that whenever the optimizer needs to read the cells (steps) of a histogram for costing, it must first place them into procedure cache. Once the query is complete, the cells are removed from cache. The amount of

cache used for each cell is minimal (two to eight bytes depending on the datatype). The reading of the cells into cache can indeed add a little time to the overall parse and compile time, but usually only a few milliseconds. These milliseconds are usually a good tradeoff for the more efficient plans that can be generated from the more granular statistics.

Don’t be concerned about having a few hundred steps in a histogram (particularly one inherited from an old distribution page during upgrade). You won’t notice the difference in parse and compile times, and the optimizer will have more statistics to work with. If you’re tempted to request tens or hundreds of thousands of steps, you may want to reconsider why you need this many. The bottom line here is not to be too concerned; there are far more important things to think about when tuning.

#### Some Optimizer-Related FAQs

##### 1. What is data skew and how can it effect the optimizer?

Data skew is a common term used to describe a column with values that are highly duplicated and other values that are not. In ASE 11.9.2 and above, the presence of both Frequency count and Range cells in the column’s histogram indicates some degree of data skew. If all cells in the histogram are one type or the other, there is no data skew in the statistics. The column may be dense, but it’s not skewed.

Is date skew bad? No, not really. It’s a normal state of many datasets. As far as the cells in the histogram and thus the costing of SARGs are concerned, Frequency count cells are desirable. Because a Frequency count cell represents only one value, the weight of the cell measures exactly how many rows in the column are occupied by that value.

Data skew does have an effect on the column’s total density value, which is used to estimate the cost of joins of the column. The total density value is created by using a geometric averaging method. One effect of this is that highly duplicated values in Frequency count cells have a proportionally greater effect on the total density value than those values that are not highly duplicated. Thus, values that may not be participating in the join have an effect on how the optimizer estimates the cost of the join.

By using optdiag, there are a couple of ways to check the distribution of values in a column and measure any data skew that may be present. The first is to compare the Total and Range cell density values. The Total density value is a measure of the average number of duplicates in the entire column. The Range cell density is a measure of the average number of duplicates in cells that are not Frequency count cells. If the values are the same, then there are no frequency count cells

in the histogram and thus no data skew. It is possible that the column may have some degree of data skew, but if there are not enough cells (steps) in the histogram to allow frequency count cells to appear, it will not appear in the statistics. If data skew doesn't appear in the statistics, it won't effect costing.

In this first piece of optdiag output, the two density values are the same. There is no data skew in this histogram:

```

Statistics for column:      "col_A"
Last update of column statistics: Dec 11 2000 11:37:693PM

Range cell density:      0.0000576586096924
Total density:          0.0000576586096924

```

Here the degree of data skew is due to the large difference between the two density values. Also notice that there is a large frequency count cell for value 1. In this case, it's responsible for the skew.

```

Statistics for column:      "col_B"
Last update of column statistics: Nov 30 2000 1:58:00:906PM

Range cell density:      0.0381101996119019
Total density:          0.6521914581601986

Step Weight      Value
1 0.00000000    < 1
2 0.79821283    = 1
3 0.10293456    < 2
4 0.00565034    < 3

```

Here there is some data skew, but the degree is low. There were two frequency count cells with relatively low weights.

```

Statistics for column:      "col_C"
Last update of column statistics: Nov 30 2000 1:57:47:890PM

Range cell density:      0.0086235950083850
Total density:          0.0762508697619250

Step Weight      Value
1 0.00000000    < 16
2 0.02647445    < 32
3 0.10705626    = 32
4 0.00250300    < 33
5 0.21902847    < 34
6 0.04812700    < 60

```

What to do about data skew? See the Q3 2000 issue of the *ISUG Technical Journal* in my article, "Where, Why and How To Add Or Modify Optimizer Statistics."

## 2. Why Does Update Index Statistics table\_name Take Longer Than update statistics table\_name?

Let's take a look at the behavior of the "standard" syntax first: **update statistics table\_name [index\_name]**. In pre-11.9.2 versions of ASE, statistics were an attribute of an index and could only exist for the leading column. When **update statistics** was run, the index was read and the statistics gathered from there. This was done because the index is sorted in the leading column order.

In 11.9.2 and above, the column-level statistics are no longer associated with an index, but are now an attribute of a column. This means that they can exist on any column whether or not it is a member of an index.

In 11.9.2 and above, if you run **update statistics table\_name**, it will behave the same as in earlier versions: it will scan the index to gather the statistics because the rows are in order. In 11.9.2 and above, **update statistics** has been extended to allow you to place statistics on columns other than the leading column of an index. There are three ways to do this:

- ◆ **update statistics table\_name (column\_name)**  
Will create or update statistics on the specified column
- ◆ **update index statistics table\_name [index\_name]**  
Will create or update statistics on all columns of all indexes in the table or the specified index.
- ◆ **update all statistics table\_name**  
Will create or update statistics on all columns of the table, and updates partition statistics.

When the column that is having statistics either created or updated is not the leading column of an index, the values must be read from the column and sorted before statistics can be gathered. A table scan is done to read the values from the column. The sort will be done in a worktable in tempdb. Both of these, of course, add I/O to the process, and this is the bulk of the extra work that needs to be done.

If you use **update index statistics**, you may want to increase the size of tempdb. This is especially true if you use **update all statistics**. In most cases, the more efficient plans generated by the optimizer will outweigh the added maintenance cost (except possibly that of **update all statistics**).

### 3. What is the *formatid* column in *sysstatistics*, and what do the values in the column mean?

The *formatid* column in *sysstatistics* holds a value that describes what type of statistical value is stored in the rest of the row (columns *c0-c79*).

- ◆ *formatid* 100: These are values that relate to the column. They include Total density, Range cell density, the number of requested and actual steps used in the histogram, the default selectivity values, the column datatype, length, scale and precision.
- ◆ *formatid* 102: The histogram's boundary values
- ◆ *formatid* 104: The histogram's cell weight values
- ◆ *formatid* 20: These are the simulated table/index level statistics that will be read when *set statistics simulate* is on, and simulated statistics exist.
- ◆ *formatid* 40: These are the simulated server configurations (cache sizes, total memory, max parallel degree, etc.) that will be read when *set statistics simulate* is on, and simulated statistics exist.

While *formatid* is a tinyint datatype, the columns that contain the values (*c0-c79*) are all varbinary. Keep this in mind if you want to query the table.

### 4. How can I tell how many steps were used, or how many steps were requested, for a column and the last time statistics were modified without having to run *optdiag*?

It's not always possible or practical to use *optdiag* to see the statistics. Also, if you're interested in automating things, *optdiag* is not going to be terribly useful. If you can't or don't want to use *optdiag* to determine the requested and actual step count, the query below will get the information for you. It will return the table name, the column name the number of actual steps used in the column's histogram, and the date of the last modification to the statistics.

```
select object_name(s.id) tablename,
       c.name column_name,
       convert(int, c4) actual_steps,
       convert(int, c5) requested_steps,
       moddate
from sysstatistics s, syscolumns c
where formatid = 100
and s.id = c.id
and colid = convert(tinyint, substring(colidarray,1,1))
and s.id > 100
and s.c4 != NULL
and s.c5 != NULL
order by 1,2
```

### 5. How can I Read The Cluster Ratios Without Going to *Optdiag*?

The cluster ratio values you see in *optdiag* output under "Derived statistics" are values that have been computed internally. The exact formula for doing this is not publicly available. However, there is a way to view the cluster ratios without going to *optdiag*.

The "raw" numbers, or CR counts, that are used to derive the cluster ratios are stored in *systabstats*. Think of each as a count of "jumps." For the Data Page and Index Page Cluster Ratio, the measure is how many jumps between extents will have to be done in order to read the data or index pages in order. For the Data Row Cluster Ratio, the value is measuring how many jumps between data pages will need to be done to read rows in index leaf order.

While the database is static, run **update statistics**. Then run the query below. *sp\_flushstats* will copy the most recent table/index level statistics from the in-memory copy to *systabstats*. In most cases, you won't need to run *sp\_flushstats*, because Housekeeper will flush the values to *systabstats* as part of its regular work. This will give you a baseline to use when monitoring the values. From time to time, rerun the script below. As the values change, so do the cluster ratios. If the values increase the cluster ratio decreases (becomes less clustered) and vice versa if they decrease.

Note: If you want to get the table level CR counts and either the table is DOL or is APL and doesn't have a clustered index, drop the clause **and t.indid != 0**.

```
sp_flushstats table_name
go
select
i.name,
t.dpagecrcnt "DPCR Raw",
t.ipagecrcnt "IPCR Raw",
t.drowcrcnt "DRCR Raw"
from sysindexes i, systabstats t
where t.id = object_id("table_name")
and t.id = i.id
and t.indid = i.indid
and t.indid != 0
go
```



# Using Optdiag Simulate In 'What-If' Analysis

By Eric Miner

*Easing the process of doing what if analyses and supplying datasets using Optdiag's simulate statistics mode.*

Have you ever wondered how a query would optimize if your tables were ten times their current size? Perhaps you are building a new application and wonder how useful an index will be as your data grows and changes. Suppose your boss wants you to justify your request for a full 5GB of memory. Or maybe you want to see how table fragmentation or data skew will affect your most common queries. Perhaps Sybase's technical support group has asked for the dataset to reproduce your optimizer problem?

Prior to ASE 11.9.2, doing what-if analyses and supplying datasets were not easy tasks. For what-if analyses, you had to create and then modify datasets to suit your tests. This took valuable time and resources. *Optdiag simulate* performs both tasks quickly and easily without having to change your actual dataset or send it to Technical Support.

## What Is Optdiag Simulate?

Optimizer statistics have changed dramatically in ASE 11.9.2. They are now stored in two system tables rather than being in a single distribution page and other locations. However, while easier to access, their varbinary format makes them difficult to read or write to. But *optdiag* allows you to do both easily. It outputs the statistics in three formats: ASCII, binary, and *simulate*. While each has its purpose, this article focuses on *optdiag simulate*.

*Optdiag's simulate statistics* mode has two functions—to perform what-if analysis, and to help Technical Support repro-

duce your optimizer cases. Either way, the actual data is not required. This saves you a great deal of effort. In *simulate* mode, *optdiag* outputs a file containing the same information as other *optdiag* modes, plus information about certain server configuration values.

For what-if analysis you can run queries against various statistics to see how different statistics affect query plans. Keep statistics as close as possible to the values you expect in the actual dataset. For example, if you only have 500MB of memory available, it may not be very useful to simulate the use of 1GB memory.

## What Can And Cannot Be Simulated

You can simulate all table and index statistics, along with cache sizes, configured degrees of max parallelism, and the largest partition size. Column statistics can also be changed using *optdiag simulate*; however, they will be written directly to the *sysstatistics* system table. Special steps are needed to return column statistics to their original values (see *Simulating Column Statistics and Removing Simulated Statistics*, below).

The *optdiag simulate* statistics output files contain all values that you can simulate or change. Most are obtained from the two system tables *sysstatistics* (column statistics) and *systabstats* (table/index statistics). Other values, obtained elsewhere in the server, include available caches and their sizes, information on parallelism, and the size of the largest partition.



*Eric Miner has been with Sybase since 1992, working with the optimizer team in engineering, as well as focusing on optimizer issues for technical support and product support engineering. He can be reached at [eric.miner@sybase.com](mailto:eric.miner@sybase.com).*

## Optdiag Syntax

```
optdiag simulate statistics db_name -U -P -S -o output_filename
```

Outputs one file containing simulated statistics for all tables in the database.

```
optdiag simulate statistics db_name..table_name -U -P -S -o output_filename
```

Outputs a file containing simulated statistics for the specified table.

```
optdiag binary simulate statistics db_name..table_name -U. -P. -S.
```

Generates the output in binary format (see below).

### Format of optdiag simulate Output

The optdiag simulate statistics output file format differs in one significant respect from other optdiag formats. It contains an “actual” and a “simulated” line for each statistics value that can be simulated. To input a file, use the `-i` option followed by the input file name. When read back into a target server, the simulated values are written to sysstatistics in a special format that the optimizer can use to estimate the cost of queries.

In the following example, the actual value, preceded with a #, will not be written to the system table. You edit only values in rows labeled “(simulated)” and/or those not beginning with a #, and change only these values. Use any text editor to edit the file.

<b>Data page count:</b>	<b>44308.000000000000 (simulated)</b>
<b># Data page count:</b>	<b>44308.000000000000 (actual)</b>

Optdiag can read and write statistics in two formats: ASCII (the default) and binary. The simplest way to view statistics is to use the parameter *statistics* alone in the optdiag command line. This outputs statistics in easy-to-read ASCII format. The parameter *statistics* must be in the optdiag command line for all formats. Use *binary* mode if precision is an issue; for example, with real, float, and double datatypes. Using binary format is optional.

As an example, consider using the binary format if you are simulating on an Intel machine and the source dataset is from a RISC machine, and datatypes may have precision issues. With simulated binary statistics, notice that the editable value is in binary (hex) format:

<b>Data page count:</b>	<b>0x0000000080a2e540 (simulated)</b>
<b># Data page count:</b>	<b>44308.000000000000 (simulated)</b>
<b># Data page count:</b>	<b>44308.000000000000 (actual)</b>



The default ASCII format is sufficient for most optdiag simulate work.

### How to Perform a Simulation

#### How Much Optdiag Simulate Output To Get?

There are two approaches to obtaining the necessary optdiag simulate outputs. You can get a single output file for the entire database:

```
optdiag simulate statistics database_name -U. -P.
```

or a separate file for each table you plan to use in the simulation:

```
optdiag simulate statistics database_name..table_name -U -P.
```

It's a good idea to keep unedited copies of any optdiag simulate files you will be working with.

When using one output file for the entire database, make sure that all tables and their indexes have been created in the test server. This is the best approach if your simulation uses queries that reference all or most of the database tables. If your queries use only a few tables, however, obtain a single file for each table. This lets you create only the desired tables and their indexes, and then read in the files individually.

#### Using Empty vs. Populated Datasets

It may be tempting to simply create empty tables and indexes and obtain optdiag simulate outputs from these. However, those output files will not include column statistics because no values exist in the empty tables. Either get optdiag output files from an existing dataset or create them manually. If you create the column level statistics manually, make sure that the statistics reflect the expected data distribution.

It is advisable to get your optdiag simulate output files from an existing dataset; this saves you the time and effort of creating the column statistics manually.

You can also cut and paste column statistics into an optdiag simulate file that was obtained from an empty dataset. To do this, the column must exist in the test server, and you must paste in the entire section of statistics for the column. Also, make sure to remove the column's name from the "No statistics for remaining columns:" list at the bottom of the output file. Column statistics will be needed if you are simulating an existing dataset or if you want to know the effects of adding statistics to columns that currently don't have them in the source server.

### Query Outputs from an Existing Dataset

If simulating with a query or queries that are usually run against an existing dataset, you should get optimizer outputs from runs on the source dataset. These output files are essential for comparison against query plans optimized using simulated statistics.

Get the following outputs from runs of your queries and/or procedures:

```
set showplan on
dbcc traceon (3604,302,310)
```

When using simulated statistics there is no need to use **set statistics io on**. If you are simulating on empty datasets, or those different in size from the source, the I/O you see may differ greatly from I/O on the source dataset. This could result in misleading I/O outputs. For example, if you are working with a 100-page table and you decide to simulate 10,000 pages, statistics I/O output will show that a table scan costs 100 I/Os, not 10,000.

Use **set statistics time on** to see differences in parse and compile time and its corresponding server CPU time. But note that the server elapsed time and total CPU time will change when the simulated values become actual values.

### Testing Queries Against Simulated Statistics

After reading in the simulated statistics via optdiag, you will want to see how they affect optimization of your queries. To do this, issue the following command (directly in isql or in a script):

```
set statistics simulate on
```

This set command tells the optimizer to use the simulated statistics stored in sysstatistics rather than the actual statistics.

If you are testing a previously compiled stored procedure, issue **set statistics simulate on**, then execute it with the recompile option. This allows the use of simulated statistics when compiling the procedure.

### Verifying that Simulated Statistics Were Used

Both Showplan and traceon 302 report when the optimizer is using simulated statistics.

Traceon 302 example:

```
Statistics for this column have been edited.
```

Showplan example:

```
Optimized using simulated statistics.
```

### Simulating Shared Statistics

The ability to change shared statistics that are not in the system tables is unique to the optdiag simulate format. Be careful to use reasonable cache, parallelism and partition size values when simulating in the test server.

### Caches

Caches from the source server that appear in the optdiag simulate output file must be present in the test server before you can use them for simulation. Otherwise, optdiag simulate will fail. You can create caches on the test server using the minimal amount of cache (512K). Also, if tables and/or indexes are bound to named caches in the source server, or if you want to simulate objects bound to caches, you must bind these objects to caches in the test server before simulating.

Here we simulate a default cache with a 16K pool of 50Mb while the test server's actual default data cache has no 16K pool.

	Size of 2K pool in Kb:	21554 (simulated)
#	Size of 2K pool in Kb:	21554 (actual)
	Size of 16K pool in Kb:	25600 (simulated)
#	Size of 16K pool in Kb:	0 (actual)

### Max Parallel Degree and Max Scan Degree

These two settings can be set as you wish for what-if analysis. If working on a case with Sybase technical support, for example, you will need to give your support engineer all information about any session level settings for these values you have used with the query(ies).

Max parallel degree:	12	(simulated)
# Max parallel degree:	1	(actual)
Max scan parallel degree:	12	(simulated)
# Max scan parallel degree:	1	(actual)

**Partitions**

For a partitioned table, the size of the largest partition appears at the end of the “statistics for table” (or clustered index if one exists) section of the optdiag output. Changing this value affects the costing of queries using parallelism. If a table is partitioned in the source server, or if you want to simulate with partitioned tables, you must partition the tables in the test server.

A table must be partitioned the same way in the test server as it is in the source server, i.e., it must have the same number of partitions.

Pages in largest partition:	5730.0000000000000000	(simulated)
# Pages in largest partition:	573.0000000000000000	(actual)

**“Simulating” Column Statistics**

Column statistics are modified the same way in all optdiag modes. When you change column statistics via optdiag simulate, they are written directly to sysstatistics, and remain in place until overwritten. They are not simulated as are other statistics. Note that in the optdiag simulate output file, the column statistics do not have an “actual” and “simulated” line for each value. Instead, all values can be edited. The optimizer uses any changes to column statistics for all queries, simulated or not.

To set column statistics back to their original values, either run update statistics on the table or column or read in an optdiag file obtained before modifying the column statistics. See Removing Simulated Statistics below more information.

Changing the column statistics enables you to simulate different data distributions, densities and default selectivity values. These values will have a direct, and often dramatic, effect on the optimizer’s query plan choice.

**Changing and Removing Simulated Statistics**

During simulation, you may want to return the simulated statistics to their original values, or change them to different values. To remove the shared statistics, use the following command from within the server:



```
delete shared statistics
```

This removes the shared statistics from system tables in the master database; the actual values will be used for the shared statistics. Shared statistics are neither columns nor tables/indexes, but are statistics used throughout the server. They include the degree of parallelism and information on available caches (cache size and I/O size).

**To Return to Original Column Statistics**

There are two ways to return to the original values after your simulation.

1. Use the **delete statistics table\_name** command for each table. This removes all column statistics for the table. Next, run **update statistics table\_name** to create statistics on all index leading columns. If other columns have statistics, run **update statistics column\_name** to create statistics on those columns. Since this could be slow for a large table, you may want to consider the second approach below.
2. Read in the original, unedited optdiag simulate output file you obtained before beginning your simulation. This will revert the column statistics to their original values.

**To Return to Original Table and Index Statistics**

To use the original table and index statistics, simply run **set statistics simulate off**. The table and index simulated statistics will be used until you run **set statistics simulate off**.

**To Change Statistics**

To change simulated table or index statistics or the column statistics, input an optdiag simulate output file. Step 2 above is the best approach for replacing simulated statistics values.

## Example: Changing The Cache Values—Simulating A 16K I/O Pool

The following example is a fairly dramatic demonstration of optdiag simulate. Here we simulate the presence of a 16K I/O pool in the default data cache. There is no 16K pool, but we simulate a 50Mb pool. First, we run the query without the simulated 16K pool. The traceon 302 and 310, along with the showplan output, indicate that only 2K I/O was used. Note: portions of the outputs have been removed to save space.

A very simple query:

```
1> select count(*)
2> from block
3> where order > 94405
4> and part >= 14745
5> and shipdate >= "6/1/98"
6> go
```

### DBCC Traceon 302 Output:

```
The best qualifying index is 'block_order_nc2' (indid 5)
costing 9300 pages,
with an estimate of 9135 rows to be returned per scan of the table,
using no index prefetch (size 2K I/O) on leaf pages,
in index cache 'default data cache' (cacheid 0) with MRU replacement
using no data prefetch (size 2K I/O),
in data cache 'default data cache' (cacheid 0) with MRU replacement
Search argument selectivity is 0.045673.
```

\*\*\*\*\*

### DBCC Traceon 310 Output:

#### CACHE USED BY THIS PLAN:

CacheID = 0: (2K) 2 (4K) 0 (8K) 0 (16K) 0

FINAL PLAN (total cost = 177918):

Notice the high estimated cost for the query.

### Showplan Output:

#### QUERY PLAN FOR STATEMENT 1 (at line 1).

##### STEP 1

Using I/O Size 2 Kbytes for index leaf pages.  
 With MRU Buffer Replacement Strategy for index leaf pages.  
 Using I/O Size 2 Kbytes for data pages.  
 With MRU Buffer Replacement Strategy for data pages.

Now we simulate a 50 Mb pool and rerun the query:

Configuration for cache:	"default data cache"
Last update of simulated:	Apr 12 1999 10:39AM
Size of 2K pool in Kb:	21554 (simulated)
# Size of 2K pool in Kb:	21554 (actual)
Size of 16K pool in Kb:	25600 (simulated)
# Size of 16K pool in Kb:	0 (actual)

### DBCC Traceon 302 Output:

Statistics for this column have been edited.

```
The best qualifying index is 'block_order_nc2' (indid 5)
costing 9300 pages,
with an estimate of 9135 rows to be returned per scan of the table,
using index prefetch (size 16K I/O) on leaf pages,
in index cache 'default data cache' (cacheid 0) with MRU replacement
using data prefetch (size 16K I/O),
in data cache 'default data cache' (cacheid 0) with MRU replacement
Search argument selectivity is 0.045673.
```

\*\*\*\*\*

### DBCC Traceon 310 Output:

#### CACHE USED BY THIS PLAN:

CacheID = 0: (2K) 0 (4K) 0 (8K) 0 (16K) 16

FINAL PLAN (total cost = 51000):

Notice that the total estimated cost of this query plan was considerably higher when only 2K I/O was available. In this case, there may be good justification for using 16K I/O.

### Showplan Output:

#### QUERY PLAN FOR STATEMENT 1 (at line 1).

##### Optimized using simulated statistics.

##### STEP 1

Using I/O Size 16 Kbytes for index leaf pages.  
 With MRU Buffer Replacement Strategy for index leaf pages.  
 Using I/O Size 16 Kbytes for data pages.  
 With MRU Buffer Replacement Strategy for data pages.

# An Introduction to Utilizing Optdiag Output

By Eric Miner

*Helping users understand output from ASE 11.9.2's new optdiag tool*



Eric Miner has been with Sybase since 1992, working with the optimizer team in engineering, as well as focusing on optimizer issues for technical and product support engineering. He can be reached at [eric.miner@sybase.com](mailto:eric.miner@sybase.com).

The optdiag utility is now being shipped with ASE 11.9.2 and above. Optdiag is an extremely useful tool that enables the user to read and monitor statistics values used by the optimizer and determine when it is best to reorg a table. You can also use it to write the column level statistics directly.

In an earlier article (3rd Quarter, 1999), I discussed how to use optdiag's *simulate statistics* mode in what-if analysis. In this article I'll step back a bit and take a look at general optdiag output and how you can use its information.

As you may have read in the documentation or here in the *Journal*, optimizer statistics are now stored in two system tables. *Systabstats* holds the table and index level statistics such as page count, row count, index page count, and the new cluster ratios. *Sysstatistics* holds the column level statistics, known as the distribution statistics, such as density values and histogram values. These statistics were formerly stored on the distribution page of each index. New to the column level statistics are the default selectivity values (replacing the "magic number"), which you can replace with any value you want. The date and time of the last change to the column level statistics is also a new value.

This article will look at some of the more useful values at the both table/index (a smaller set of values that do not depend on the data itself) and column level statistics which describe the distribution of the data. All of the statistics in optdiag are used by the optimizer to estimate the cost of a query. Some are more important to you on a daily basis than others. Some you can use to monitor table fragmentation, and others to see the distribution of values in your dataset.

## Table/Index Level Statistics

The table/index level statistics describe the table and its indexes to the optimizer. These statistics are stored in *systabstats* and are dynamically maintained by ASE (note that **update statistics** still needs to be run to keep the column level statistics up to date). They are not wholly new in 11.9.2; some, such as row counts and page counts, were stored in various structures within ASE, i.e., the OAM page. The cluster ratios are new, however; only hard-coded "magic numbers" were previously used to include clustering in the cost estimate.

As changes are made to the table and its indexes, these values are updated in memory. The in-memory values are then flushed to *systabstats* in a number of ways. The table/index level statistics do not generally need to be manually flushed to

*systabstats*, as Housekeeper will do this as part of its regular routine. If you believe that the table/index level statistics are not up-to-date, however, the following actions will flush them from cache to *systabstats*, including [link to existing text] including **shutdown**, **update statistics**, **running optdiag**, **sp\_flushstats**, **dbcc flushstats**, and **checkpoint**. Because these statistics are often flushed from memory, they cannot be overwritten using *optdiag*.

The table/index statistics are used by the optimizer, along with the column level statistics, to estimate the cost of accessing the required data. For example, the number of rows and pages in a table is a fundamental component of costing a query. It would be hard to know if an index access is the cheapest method if the cost of a table scan was not available for comparison.

The new cluster ratios are used to estimate the cost of large I/O access and non-clustered index access.

**Table Statistics**

The first statistics example below is taken from an All Page Lock (APL) table with a clustered index. Here the statistics are for the table only. In the second example, we're looking at a Data Only Lock (DOL) table with a clustered index. In this case, the statistics listed in this section are for the clustered index. For the APL Table:

<b>Table owner:</b>	"dbo"
<b>Table name:</b>	"table_1"
<b>Statistics for table:</b>	"table_1"

For the DOL Table:

<b>Statistics for index:</b>	"clustered_index_name"
------------------------------	------------------------

The name and type of the index. This can be useful to verify you're looking at the right index and to quickly identify what type it is. Remember, that if you have a clustered index on a DOL table, its structure will be that of a non-clustered index.

<b>Data page count:</b>	44308
-------------------------	-------

This is an important value to the optimizer, as it is used in all cost estimates. As the number of pages grows, there is a chance that the optimizer may find that a previously efficient query plan now is not so efficient. If it is estimated that a table scan will cost less I/O than an index access, the table

scan will be chosen. There is no set ratio used to determine efficiency; the optimizer will depend on the estimated cost of an individual query.

<b>Empty data page count:</b>	1
-------------------------------	---

The number of empty data pages in the table. These are pages that have no rows but are allocated to extents with pages that are not empty. Keep an eye on this value; as it increases, you may want to consider running **reorg**. Empty pages are included in the total page count and can thus affect the optimizer's cost estimates. Empty pages are also included when the cluster ratios are calculated.

<b>Data row count:</b>	600572.000000000000000000
------------------------	---------------------------

A very important value to the optimizer: The number of rows in the table is a major part of estimating costs. Keep an eye on this value. Remember from time to time this value may not exactly reflect what's in the table. It may take a little time for it to be flushed from memory (this is true of all values stored in *systabstats*).

<b>Forwarded row count:</b>	0.000000000000000000
-----------------------------	----------------------

The number of rows that have moved from their original page to another page. A pointer to their current page is maintained in the original page. If the number of forwarded rows increases, you may want to consider running **reorg**. The number of forwarded rows is included when the cluster ratios are calculated.

<b>Deleted row count:</b>	0.000000000000000000
---------------------------	----------------------

The number of committed deletes whose space has not yet been reclaimed. As this number increases, so does table fragmentation. This could result in the same number of rows being on more pages, and this in turn can affect optimization.

<b>Data page CR count:</b>	5539.000000000000000000
----------------------------	-------------------------

The "raw number" used to derive the Data Page Cluster Ratio. Wherever you see "CR count," it is the number used to derive the cluster ratio. This also applies to both table and index statistics.

<b>Derived statistics:</b>	
<b>Data page cluster ratio:</b>	0.999990000000000001

The cluster ratio derived from the data page “CR count,” above. This is the value the optimizer uses when estimating the cost of accessing the table using large I/O. It’s a measure of how many I/Os will be required to read the table, and thus shows how well pages of an APL table are clustered on the extents. For a DOL table, it measures how well packed pages are on extents (fewer empty pages increases packing). The closer it is to 1, the better the clustering.

### Index Statistics

These statistics apply to non-clustered indexes on both DOL and APL tables and to all indexes on DOL tables.

Statistics for index:	"table_1_lorder" (nonclustered)
Index column list:	"l_orderkey"

A listing of all columns in the index, which can be very useful information when you want to quickly identify columns in the index.

Leaf count:	2464
-------------	------

The number of leaf level pages in the index. For example, the cost of scanning this entire non-clustered index would be the height of the index plus the number of leaf pages.

Empty leaf page count:	0
------------------------	---

The number of empty leaf pages in the index. These are essentially the same as empty data pages except in the index leaf. Any empty pages are counted towards the total page count by the optimizer and are a waste both of space and page count. If the number increases, consider dropping and recreating the index or running reorg.

Data page CR count:	5811.0000000000000000
Index page CR count:	311.0000000000000000
Data row CR count:	44579.0000000000000000

The raw numbers used to get the derived statistics (cluster ratios), discussed in detail below.

Index height:	2
---------------	---

The height of the index not including the leaf level. This value is included by the optimizer whenever it estimates an access of the index.

### Derived Statistics

Derived statistics are the cluster ratio values used by the optimizer in costing, and are derived from the CR counts. The closer to 1 the value is, the better it is clustered. Note that table fragmentation will affect the data and index page cluster ratios.

Data page cluster ratio:	0.9930327868852459
--------------------------	--------------------

Used to estimate the cost of large I/O data page accesses using this index. It’s the same as the table’s DPCR, except that this value relates to the index.

Index page cluster ratio:	0.9986085343228200
---------------------------	--------------------

Used by the optimizer to estimate the cost of reading a large number of index leaf pages using large I/O; for example, a covered index access. It measures how well index pages are clustered in relation to extents. This value is used when costing large I/O for non-clustered indexes and all indexes on a DOL table.

Data row cluster ratio:	0.9995110244218134
-------------------------	--------------------

A measure of how well clustered data rows are on data pages, in relation to rows in the leaf level of this index. You’ll notice that this number varies considerably between indexes. For example, if the index is clustered (on an APL table or a placement index on an DOL table), this value will be 1, or very close to it. If the index is built on other columns, it may be very low.

Another way to think of this value is that it indicates how much jumping around a read will have to do in the data pages, in order to read row sequentially from the leaf of the index. In pre-11.9.2 versions of ASE, there was a built-in assumption that for every read of a data row from the leaf of a non-clustered index, one I/O would have to be done. This was not always accurate, since sequential rows may be on the data page. The DRPCR now allows the optimizer to estimate essentially how efficient a non-clustered access will be.

### Column Level Statistics

Column level statistics describe aspects of the data itself to the optimizer. They consist of information about the distribution and density of values in the column. They also contain the date and time of the last modification to the statistics. No more wondering when the last time **update statistics** was run!

When should **update statistics** on a column be run? This is the age-old question, and one that does not have a simple answer. The best answer is that “it depends on your data and how it changes.” The old rule of thumb is to run **update statistics** when 5-10% of the rows have been changed.

A major revision to the ASE distribution statistics is that they are now associated with a column, rather than with an index as in earlier versions. Any column you choose can have statistics on it. Unlike the table/index level statistics, any of the column level statistics can be changed by the user via **optdiag**.

<b>Statistics for column:</b>	<b>"l_orderkey"</b>
-------------------------------	---------------------

This is the name of the column whose statistics are listed below.

<b>Last update of column statistics:</b>	<b>Sep 22 1999 2:50:38:703PM</b>
--	----------------------------------

The date and time that the last modification of the statistics occurred. This can be due to create index, update statistics, or by writing the statistics with **optdiag**.

### A Word About Densities

The density values are a measure of the number of duplicate values in the column. However, they each measure a different aspect of the column. The closer to 1, the denser the values; that is, the more highly duplicated the values are. These values are overwritten when you run **update statistics**. You'll have to input an **optdiag** file afterward to set them back to the values you want (such as in cases of data skew).

<b>Range cell density:</b>	<b>0.0397305175490779</b>
----------------------------	---------------------------

A measure of the density of values in the column, after values in frequency count cells have been eliminated (more on cell types below). Basically, this is the density of all values that are not highly duplicated. This value is used to estimate the cost of an equi-SARG when the value falls into a range cell. If the range cell and total density values match, there are no frequency count cells in the column.

<b>Total density:</b>	<b>0.0408260000000000</b>
-----------------------	---------------------------

A measure of the duplicate values in all rows of the column, no matter what cell type they fall into. This value is used when estimating the cost of a join against the column. It is also used as the selectivity value when the value of an equi-SARG is not known at run time.

### The Default Selectivity Values

The two default selectivity values replace the “magic numbers” that were previously hard-coded into ASE. These are used when the SARG value is not known at runtime, such as when a local variable is defined and a selected value is used in a SARG. Missing here is the default selectivity for an equi-SARG (as mentioned above, the total density value is used). In previous versions of ASE, a hard-coded value of 0.10 was used.

Users can change the two listed default selectivity values via **optdiag**. Of course, you'll need to run a test to determine the best one. Changing these values will have no effect on the optimizer if SARG values are known at runtime. If you do decide to change these values, be sure to make the value a number between 1 and 0 and remove the text. For example, change:

<b>Range selectivity:</b>	<b>default used (0.33)</b>
---------------------------	----------------------------

to:

<b>Range selectivity:</b>	<b>0.40</b>
---------------------------	-------------

These values will not be overwritten when you run **update statistics**.

<b>Range selectivity:</b>	<b>default used (0.33)</b>
---------------------------	----------------------------

This is the default value used for an open ended range SARG (<,<=,>,>=) when the SARG value is not known at runtime.

<b>In between selectivity:</b>	<b>default used (0.25)</b>
--------------------------------	----------------------------

This is the default value used for a between SARG (between, col >= X and col <= Y) when the SARG value is not known at runtime.

### The Histogram Values

The histogram describes the distribution of values within the column. Unlike previous versions of ASE, the histogram now contains a value for the *weight* of a cell.

#### Steps and Cells

A *step* should be thought of as the point within the data from where a value is read in order to establish a boundary value, as seen in the “Value” column of the histogram below. A *cell* can be thought of as all values that fall within the value of the step and the previous step value. A cell is inclusive of its upper boundary (the value in the step’s “Value” column) and is exclusive of its lower boundary value. A cell’s lower boundary value is one significant bit greater than the “Value” of the previous step.

With an integer datatype, this is easy to understand. If the value of a specific cell is 12 and the value of the previous cell is 6, the lower boundary of the cell you are looking at is 7. With char datatypes this is not as easy to read, because the next greater bit may be unreadable. The weight value is associated with a cell, not a step. There are two types of cells that can appear in the histogram, range cells and frequency count cells (see next page).

**Histogram for column:** "l\_orderkey"

The name of the column the histogram belongs to. This is important to check since statistics belong both to columns and to indexes.

**Column datatype:** integer

This is the datatype of the column.

**Requested step count:** 20  
**Actual step count:** 20

These values represent the number of steps that were requested and the number that were created. At times there may be fewer steps than requested via **create index** or **update statistics**. This is because the histogram could be created with fewer cells. Usually this is due to values that are highly duplicated.

Step	Weight	Value
1	0.00000000	<= 0
2	0.05263400	<= 26151
3	0.05263400	<= 52452
4	0.05263000	<= 78818
5	0.05263000	<= 104804
6	0.05263000	<= 131042
7	0.05263200	<= 157409
8	0.05263000	<= 183715
9	0.05263800	<= 210083
10	0.05263000	<= 236580
11	0.05263400	<= 262978
12	0.05263400	<= 289732
13	0.05263000	<= 315908
14	0.05263400	<= 342212
15	0.05263000	<= 368674
16	0.05263400	<= 394819
17	0.05263000	<= 421184
18	0.05263800	<= 447554
19	0.05263200	<= 473447
20	0.05261600	<= 499683

The histogram itself contains the step numbers, the weights, an operator used to help indicate the type of cell, and the value of the cell. The weight value indicates the percentage of rows in the column that are occupied by values in the cell. For example, a weight of 0.050000 means that the value(s) in the cell occupy 5% of the rows in the column. The weight is very useful to determine when a value or values is becoming highly duplicated—if the weights are fairly uniform, the data is well distributed. This is a new value in the column statistics; pre-11.9.2 histograms did not have associated weights.

**The First Step and NULL Values**

The first step of the histogram is different than the others—it represents the NULL values in the column. If the weight of step 1 is 0.00000, there are no NULL values present. If the weight is any non-zero number, however, that is the percentage of the column that has NULL values. The value for the first step is always the highest value that is less than the minimum value in the column.

In previous versions of ASE, NULL values were not represented by their own cell. This often caused inaccurate costing of queries using NULL in their SARGs.

**Range Cells**

Range cells contain more than one value.

11	0.05263400	<=	262978
12	0.05263400	<=	289732

For example, let's look at the range cell designated by step 12 above. This represents all values in the column between 262979 and 289732. You'll notice that the "value" is used as well as the boundaries of the cell. Also, the lower boundary is not the value of the previous step, which is normally one byte more than the previous value: in this case, 262979.

**Frequency Count Cells**

Frequency count cells contain only one value, and usually occur where a value or values are highly duplicated in the column, such as when a value occupies more rows than the width of a cell (the width of a cell is the number of rows ASE has determined between steps). Simply put, this is the number of rows divided by the number or requested steps.

Because frequency count cells contain only one value, they are the most accurate type of cell. The optimizer knows exactly how many rows in the column are represented by the cell using the weight.

```

6 0.05164000 <= 39975
7 0.00000000 < 40000
8 0.14059000 = 40000
9 0.05265000 <= 45319
    
```

In the above example, step (cell) 8 is a frequency count cell. The value 40000 occupies just about 14.06% of the column. This is more than the average. Notice how the range cells on either side of 40000 are both just slightly more than 5% of the column. The 0 value for step 7 is used to indicate that there is no need to calculate a lower boundary for step 8, since it only contains one value. You see how useful the weight can be when examining distribution of your data.

Technically, there are two types of frequency count cells, *sparse* and *dense*. The above example is a sparse frequency count cell, because the values are not sequential within the column. When values in the column are sequential, the cell will appear as:

```

5 0.04104372 <= 4
6 0.41043721 <= 5
7 0.05514809 <= 6
8 0.03963328 <= 7
    
```

In the example above, cell 6 is a frequency count cell even though its operator looks the same as the surrounding range cells.

```

No statistics for remaining columns:  "comment"
(default values used)                "commit_date"
                                       "discount"
                                       "extended_price"
                                       "line_number"
    
```

At the end of optdiag output for each table is a list of column that don't have statistics. This can be very useful when you're determining which columns have statistics and which may need them. The "default values used" message indicates that, since there are no statistics on these column, hard-coded values will be used when estimating selectivity for queries using these columns.

**Conclusion**

This article has only touched the surface of the optdiag outputs and their meaning and uses. Remember, before directly writing statistics in your production database, to make sure to perform tests to verify that changes result in efficient performance. [may need a couple of more sentences here] ■